Software of the PHENIX muon spectrometer

International Workshop on Computing for Heavy Ion Physics

Nantes, 2005/04/28

Hugo Pereira - CEA Saclay

- introduction to the PHENIX software framework
- overview of Muon software
- geometry management
- additional notes

PHENIX detector

central arm

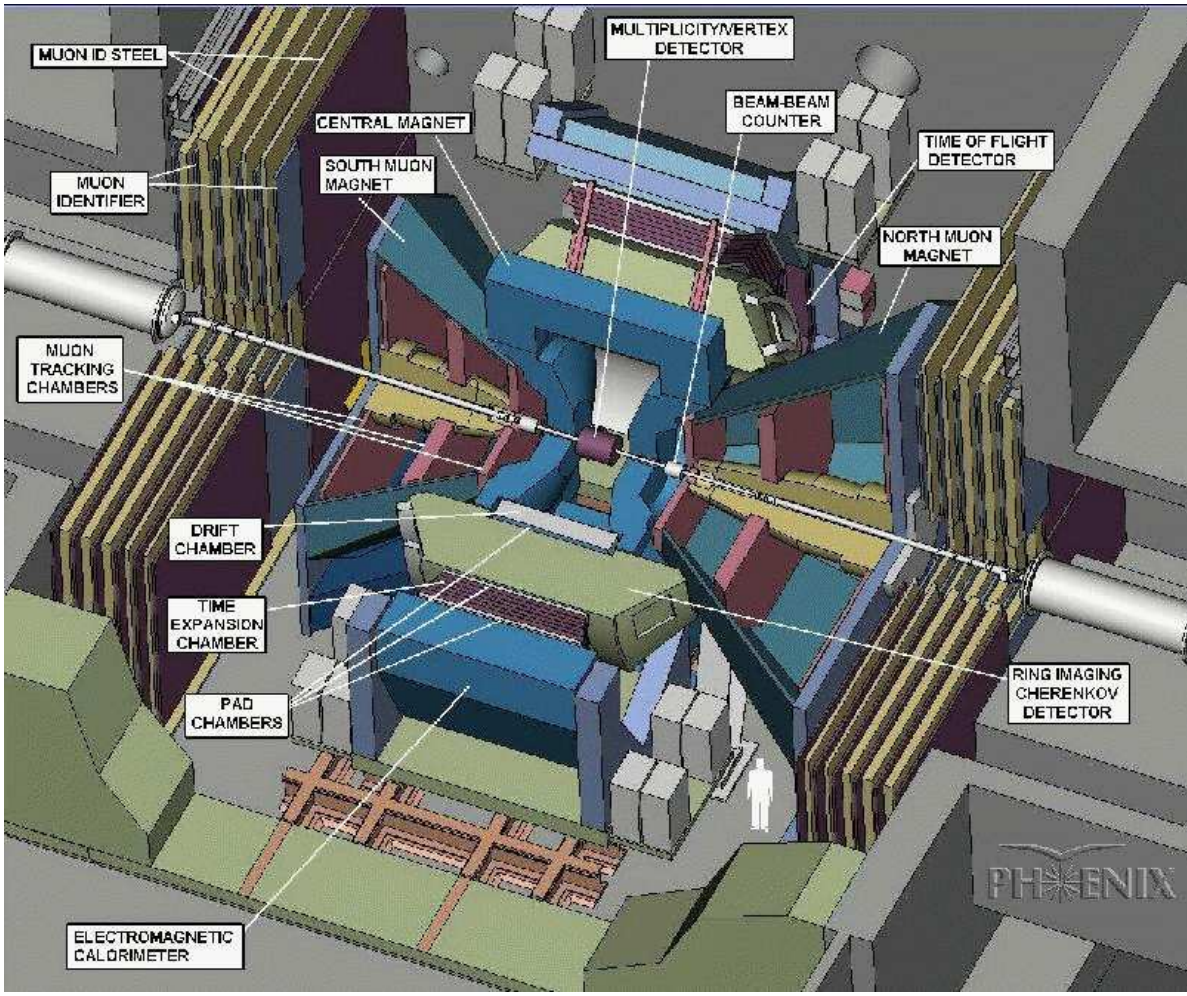
hadrons, photons and electrons

- RICH,
- pad chambers, drift chambers
- calorimeters,
- time of flight,
- etc.

muon arms

at forward rapidity

- cathode strip chambers
- Iarocci tubes



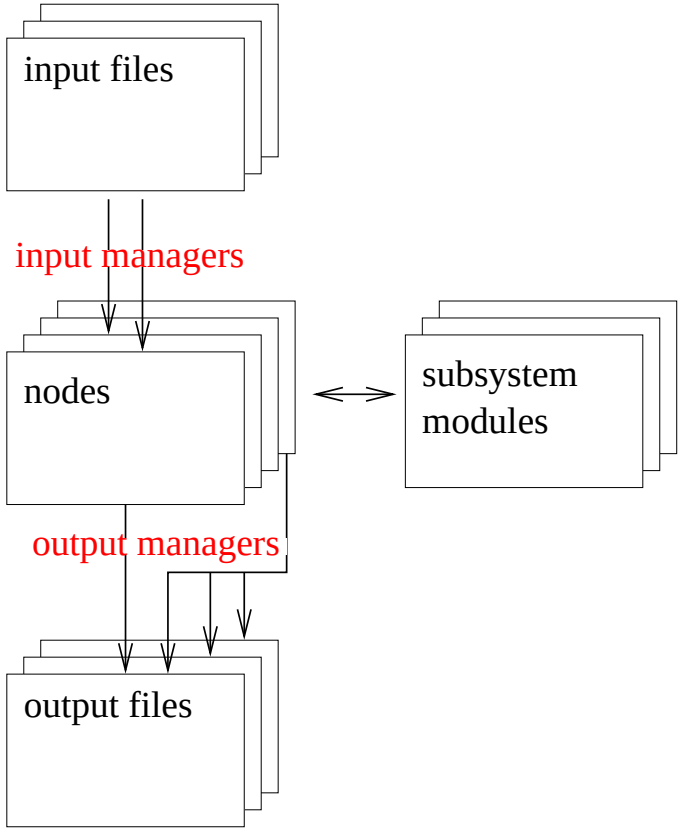
context

- a lot of different subsystems (detectors)
- different people associated
- different piece of code

software framework

a set of abstract base classes, doing (almost) nothing:

- input and output managers
- subsystem modules
- templatized tree architecture for interface objects (nodes)



Each subsystem, including the muon arms has its set of subsystem modules. They communicate only by manipulating nodes.

Muon software

Important note: almost entirely rewritten three years ago, by few people, mostly Sean Kelly, who left since then.

Previous software :

- a mixture of FORTRAN, C and C++,
- highly unstable (memory leaks, seg faults)
- but working !

New software

- officially adopted last year.
- no physics paper yet. Start with 2004 PHENIX data taking (Au+Au@200 GeV, 62 GeV and p+p@200 GeV)
- more stable, but bugs (when any) are more perverse.
- also working

has its own *framework*, embedded into dedicated subsystem modules.

has its own interface objects/containers, communicating to the *official* nodes.

Glossary [I] input/output files

PHENIX raw data files (PRDF):

- real data input files only;
- binary files generated by the DAQ; no underlying format.

DST

- both input and output, MC and real data;
- highly customizable. used for the whole pattern recognition.

nanoDST

- Output only;
- no MC information;
- used for compatibility with the rest of the (older) PHENIX software.

picoDST

- the smallest file format we have;
- must be portable, must not require loading of any *PHENIX specific* library for reading.

in the end you end up making histograms, and fitting *things* through *points*

Glossary [II]

Interface Objects

basic objects needed throughout the reconstruction (hits, clusters, tracks, etc.)

only members; no algorithm

read from/written to I/O files via nodes.

Interface Object Containers

Handle several instances of Interface Objects.

Cleared at each events.

Responsible for object persistence (that is: writing to output files)

Analysis Modules

Do the job (that is: populate/filter out the IOCs), from low level objects (real data/MC hits) to high level objects (tracks, dimuons).

Design choices

Massive inheritance/templatization

- to avoid code duplication
- to provide basic tools to all containers/modules

  basic object counting and statistics

  timers

  option driven loggers

***STL* (standard template library)** because

- is well maintained (more or less backward compatible when changing compiler version)
- has a lot of compact, optimized, build-in algorithms for sorting, finding, etc;
- is standard C++

***BOOST* for object ownership**

- BOOST smart pointers against memory leaks
- fixed sized BOOST arrays: automatically checked against overflows
- again, *almost* standard c++;

***GSL* (gnu scientific library)** for math

ROOT for I/Os, high level macros, and library loading

Interface objects evolution scheme

Evolution scheme is needed to ensure that new code still can read old files

Containers

Containers are STL maps, reading from and writing to TClonesArray (the *nodes*).

The map index is a unique key generated for each object, from its detector location + running index.

Makes it easy to retrieve subsamples of all objects (say all hits in a given detector).

Containers need no versionning because

- they store only (shared) pointers to interface base class objects.
- they are not written directly to the output files

interface objects (object written to and read from root I/O files)

object versionning uses standard c++ inheritance:

- all interface of the same type derive from a base class;
- the I/O TClonesArray stores pointers to the current (latest) version;
- all versions have a *copy* constructor from the base class,
  $\Rightarrow$ all versions can be constructed from an older/newer version.

Interface object associations

Purpose: keep track of which hits makes a cluster; which clusters makes a track; etc. Keep track of MC information to estimate efficiencies.

Main issues with object associations

- flexibility,
- reflexivity,
- persistence: keep associations in the I/O files,

the *easy* way: each interface object keeps a separate list for each type of associates.

- no flexibility: you need to modify (evolve) the object when changing the association scheme;
- object deletion is unsafe:

  when deleting one object you need to go through all the associated lists and tell each associate to remove it from there;
- handling pointers for the association is not persistent (can't be written to, read from I/O files);
- makes it difficult to disentangle MC and real data.

Interface object associations [II] our (complicated) way 11

- all interface objects derive from a Key object with a unique identifier.
- each interface object has

  a STL multimap, using the class name as a key and a (shared) pointer to the base class (Key) as a value.

  It is used to retrieve associated objects of a given type.

  a synchronized set of the corresponding unique identifiers

Using the base class solves the flexibility problem: all objects can be associated to all.

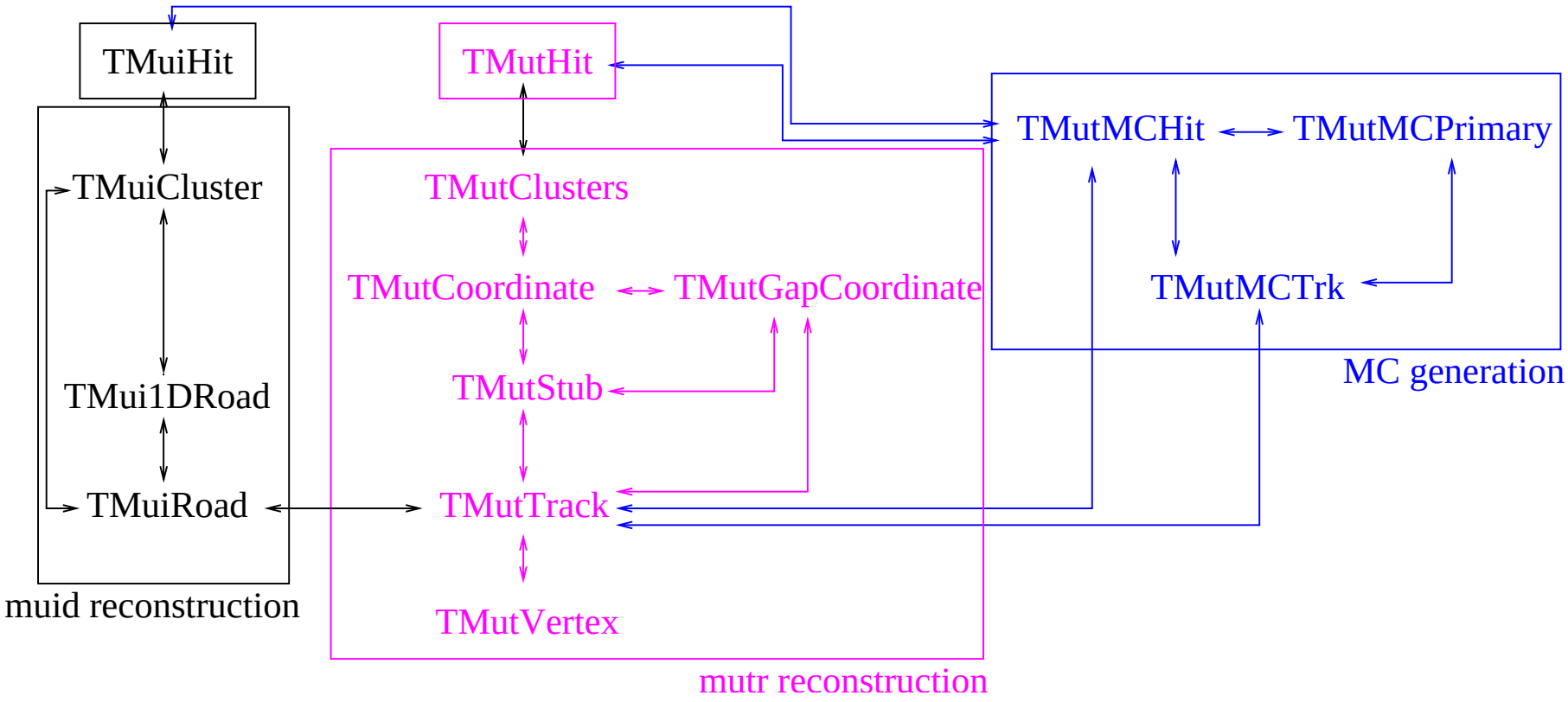
Using BOOST shared pointers solves the deletion problem (more or less).

Only the set of unique identifiers is stored to the I/O files. When reading from an I/O file, the multimap is rebuild from the unique id set. This ensures persistence.

The association technology, though quite complicated, is entirely implemented in the base class. usage within analysis modules is made **much** easier:

```
PHKey::associate( cluster_pointer, track_pointer );
track_pointer->get_associated<TMutCluster>();
```

Current association scheme



Analysis modules

Common scheme : an initialization, an event method, an end method.

Modularity :

modules don't know about each other;

Are configured/put together in proper order in the parent macro.

Communicate only via the interface objects

Three types of modules :

- unpacking modules;
- reconstruction modules;
- analysis modules.

Unpacking modules

Purpose: generate the standard *reconstructible* interface objects (hits) from either

- raw data (real data)
- monte carlo inputs (Pythia/Hijing + GEANT)
- a DST (for reprocessing).

Allow event/event embedding of (clean) MC signal into real data (messy) events
$\Rightarrow$ *realistic* background rather than Hijing

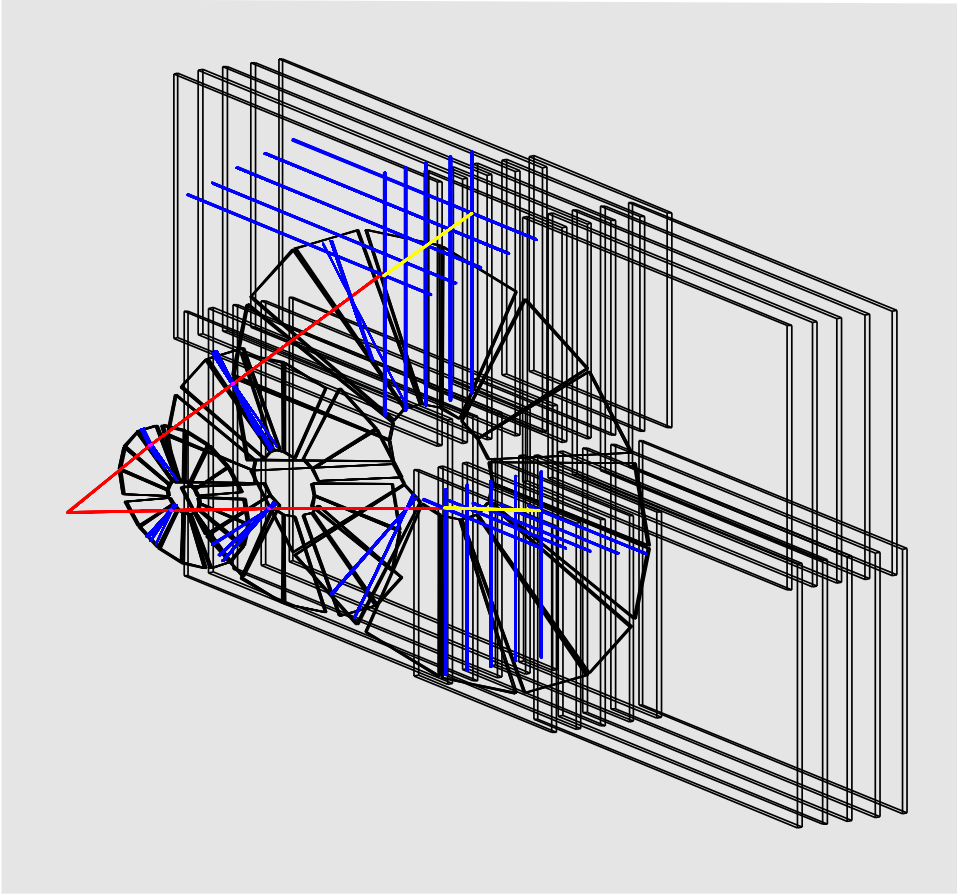
Handle decoding and calibrations (from database); detector response (on MC)

Reconstruction modules

Make the pattern recognition:

principle

- tracking starts from Muid roads
- find/fit clusters/coordinates/gap coordinates in matching octants
- find/fit stubs (track segments)
- build and fit full track
- refine track to road association to get correct depth
- fit a vertex together with BBC for each track pair



Separation between unpacking and reconstruction ensures that same algorithm runs on MC and real data.
The reconstruction do not need the detector geometry anymore.

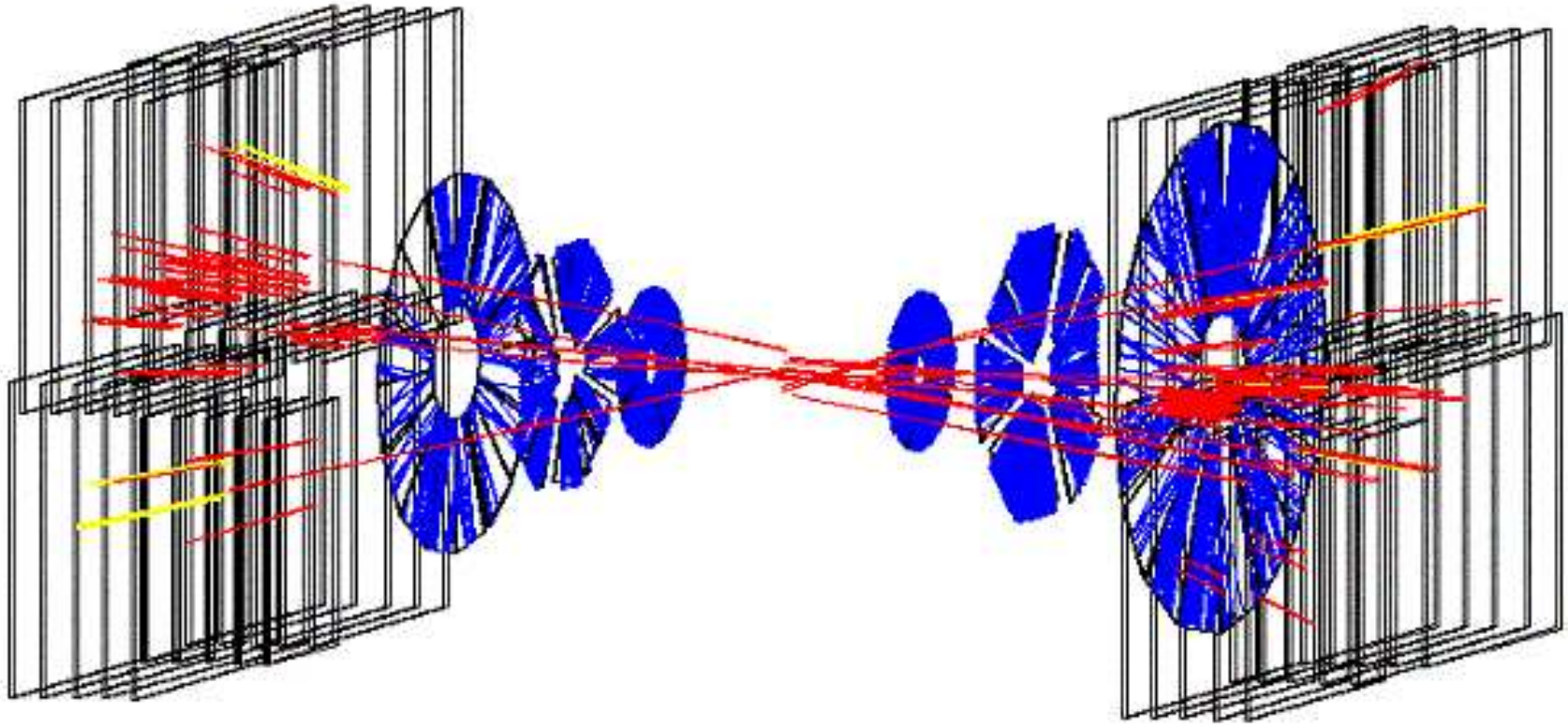
Evaluation modules

Fill regular ROOT ntuples, nanoDSTs, picoDSTs

define/apply low level cuts for

- final analysis,
- code development,
- reco vs MC matching,
- detector offline alignment
- etc ...

Event display (another evaluation module)



typical Au+Au level2 filtered event

Geometry management

Maximum modularity

- for the muon tracker, each strip is a separate object
- for the muon identifier, each tube

Three level of alignment corrections

- first level: internal strip/tubes alignment within detection planes
- second level: detection plane global alignment from Database
- third level: files for fine tuning and *realignment* procedure, before writing to DB.

Alignment tools

- survey at detector construction/installation
- optical alignment using laser and lenses (*almost* not used, but for stability checks)
- software alignment. Presently painful and done *by hand*. Work in progress to implement an automated global alignment method (similar to what's done on COMPASS; foreseen on ALICE).

Third party tools

- Why is it so slow? (profiling the code): jprof
- Why does it crash? (leak checks): valgrind, insure
- What's happening here? (automatic documentation): doxygen

Rebuild and CVS policy

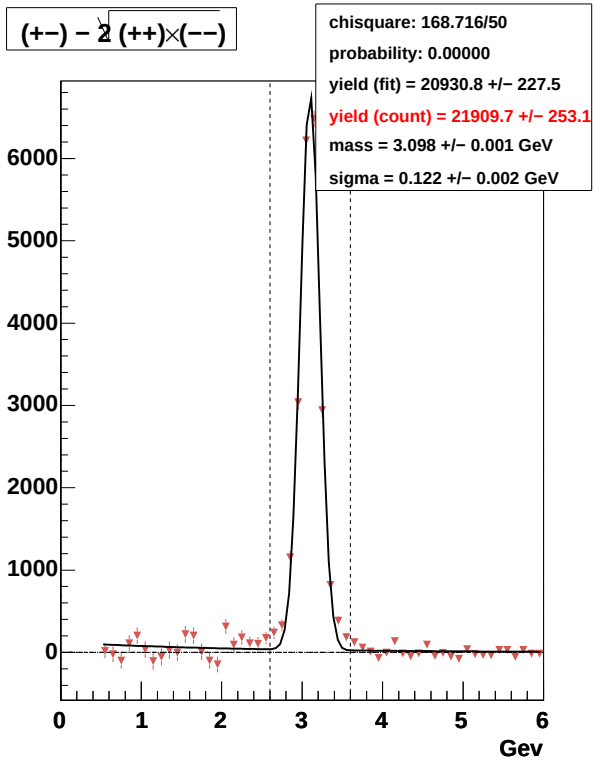
- nightly rebuilds (new version) from CVS tree;
- tagged persistent builds (pro version) for productions;
- **no CVS policy**: everybody can commit stuff. Gets an email if rebuild fails.

  pros: free and fast evolving code

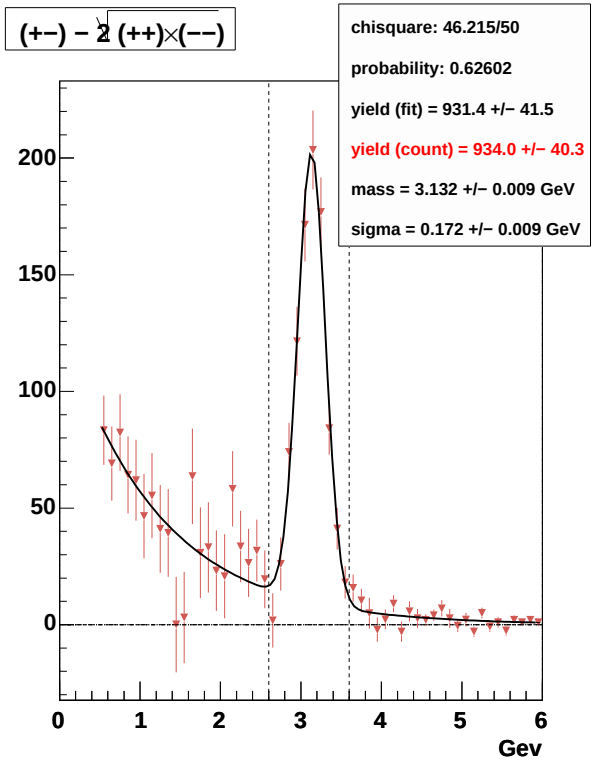
  cons: you don't teach people to write reliable code (I'm not saying I do); you may have crashed rebuilds for weeks.

This is a simplified view of the full stuff. The truth is slightly more complicated, slightly less well organized. In the end, what matters is the physics you get out of it, how fast you get it and how reliable it is.

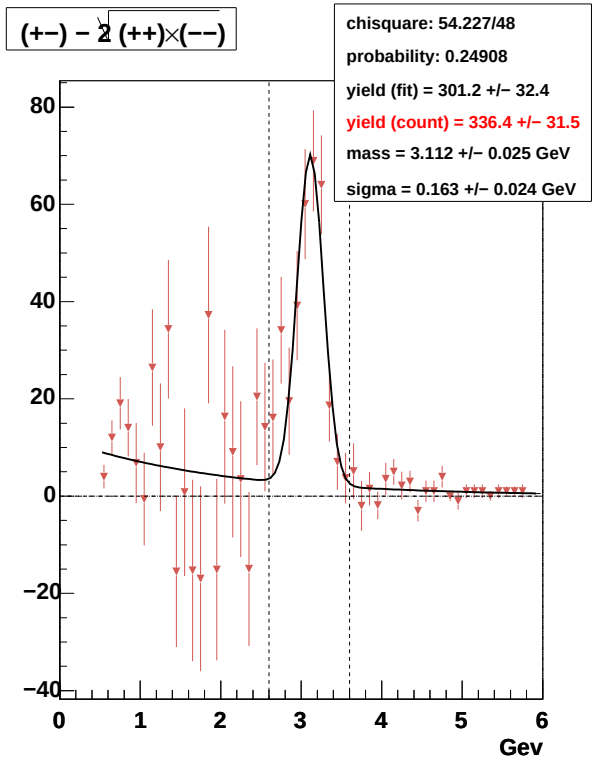
Just to say it works



Pure J/ψ MC



Run4 p+p@200GeV
(no cuts)



Run4 Au+Au@200GeV
(south arm - peripheral - official pdst

All crew busy to get results for QM05. No more software developments.

References

- STL: http://www.sgi.com/tech/stl
- BOOST: http://www.boost.org
- GSL: http://sources.redhat.com/gsl
- ROOT: http://root.cern.ch/root/Welcome.html
- DOXYGEN: http://www.stack.nl/~dimitri/doxygen/index.html
- PHENIX muon software online documentation:
  https://www.phenix.bnl.gov/WWW/publish/hpereira/doc/mutoo/html/index.html
  https://www.phenix.bnl.gov/WWW/publish/hpereira/doc/muioo/html/index.html
  https://www.phenix.bnl.gov/WWW/publish/hpereira/doc/mutoo_subsysreco/html/index.html
- misc: http://www.planettribes.com/allyourbase/AYB2.swf